A Novel Approach to Improve Concolic Testing

Arun Kumar Sahani

Roll No. : 213CS3191

under the supervision of

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769008, India

A Novel Approach to Improve Concolic Testing

Dissertation submitted in

MAY 26

to the department of

Computer Science and Engineering

of

National Institute of Technology Rourkela

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Arun Kumar Sahani

(Roll. 213CS3191)

under the supervision of

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769 008, India

Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela-769 008, India. www.nitrkl.ac.in

May 26, 2015

Certificate

This is to certify that the work in the thesis entitled *A Novel Approach to Improve Concolic Testing* by *Arun Kumar Sahani*, having roll number 213CS3191, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology* in *Computer Science and Engineering Department*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Dr. Durga Prasad Mohapatra
Associate Professor
Department of CSE
NIT, Rourkela

Acknowledgment

Firstly, I would express my profound feeling of appreciation and gratitude towards my supervisor Prof. Durga Prasad Mohapatra, who has been guiding behind this work. I need to express gratitude toward him for acquainting me with the field of MC/DC testing, Concolic Testing and issuing me the chance to work under him. His unified confidence in this subject and capacity to draw out the best of explanatory and reasonable aptitudes in individuals has been significant in intense periods. Without his precious exhortation and aid it would not have been feasible for me to finish this thesis. I am extremely obligated to him for his steady support and significant guidance in every part of my scholastic life. I think it my fortune to have got an opportunity to work with such a phenomenal person.

I thank our H.O.D. Prof. Santanu Kumar Rath for his consistent reinforcement in my thesis work. He have been incredible wellsprings of motivation to me and I express gratitude toward them in all seriousness.

I would also like to thank all faculty members of computer science department. I would also like to thank Sangharatna Godboley (Ph.d Scholar), All my Lab mantes.

Finally, I am owing debtors to my family to bolster me consistently amid from beginning of my life especially hard times.

Exceeding all I would thank to Lord "SAI BABA"

Arun Kumar Sahani

Abstract

Concolic Testing is the combination of symbolic as well as concrete execution. It considers program variables as symbolic variables along with concrete execution path. Branch Coverage belongs to white box testing. Its objective is to demonstrate that all conditions present in a predicate can impact the estimation of predicates in a particular manner. In the area of aerospace and safety critical domains, software quality assurance is strict to specified rules and regulations that maintained in DO-178 standard. To resolve such issues, Concolic testing generates automated test cases to attain high branch and MC/DC coverage in an automated technique based on the branch coverage. Here, we consider Java programs for achieving a high percentage. We are taking Java program as input that is named as J. This Java program is supplying to JPCT (Java program code transformer) which gives the output program called transformed program $J^{'}$ which is feed as input to JCUTE (Java Concolic tester tool) to generate test cases as well as to achieve more branch coverage percentage. Our study here is based on two steps, and one is taking Java program without the help of JPCT, and other is with the help of JPCT. The percentage coverage of branch is more in the transformed program without affecting the output with automatically adding some extra statements to the input. This resolves some of the bottleneck issues of the previous Concolic testing methods. We are getting the result experimentally show that our method achieves an average 17.21 percentage more branch coverage.

Keyword: Concolic Testing, MC/DC Testing, Java Program Code Transformer(JPCT), Java Concolic Tester (JCUTE)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software engineering is a systematic and cost efficient way to develop software. It is a method of development from both innovation as well as analysis of previous mistakes. Software engineering objective is to develop the quality software. It consists of many cycles with verification and validation mechanism as well as proper testing methodology. In early days of software development process software testing was applied after development of software, but now days testing links with all the phases of development.

1.1 Software Testing

Software testing is the process which identifies the bugs, errors and faults of a system to ensure a better software product. Its objective is to provide independent information about the product to the user and also the risk of failure of the product. Software testing responsible is to provide better quality software. Its aim is to test maximum number of test cases out of infinite to cover all the feasible solutions with intend to find maximum bug inside the program. It takes consideration of both normal as well as abnormal condition for the identification of unexpected errors at any condition. It is an pessimistic approach for generation of test cases .

1.1.1 Goals of Software Testing

The goals of software testing is basically divided into three different categories with so many subcategories.

1. *Instant Goal*
    - Bug Discovery
    - Bug Prevention
2. *Long Term goal*
    - Reliability Management
    - Quality Assurance
    - Customer Satisfaction
    - Risk Management
3. *Post-implementation goal*
    - Reduced maintenance cost
    - Improved testing process

1.1.2 Software Testing Life Cycle (STLC)

Software testing is a series of well-defined step wise decision to provide error less effective software. This step wise procedure is called as software testing life cycle (STLC). It mainly consists of six steps

1. *Test Planning*
2. *Test Analysis*
3. *Test Design*
4. *Test Environmental setup*

5. *Test Execution*

6. *Test Reporting*

1.1.3 Different levels of Software Testing

In the real world it is impossible to provide 100% efficient software testing.But by the help of an effective testing it is possible to afford alpine level of recruitment of testing. The way in which we are detecting the test cases is known as software testing techniques. The basic objective of test cases is to find out maximum no.of bugs and to cover large area. Testing is done everywhere of the world but it behaves differently in different circumstances There are various levels of testing exist such as Unit validation Testing, Integration Testing, Function Testing, System Testing, Acceptance Testing.

1. **Unit validation Testing:** It is the primary level of software testing where each and every little module is being tested individually.If one single error can not remove from beginning in future it will create a worthless software.

2. **Integration Testing:** After testing of individual module when they aggregated the new combined modules had been tested in a way which is called as Integration testing. Here testing applied on group of bounded modules.

3. **Function Testing:** It is used to identify the dissonance between the real behavior on the module and the functional specification. The quality of any functional module has been tested under the category.

4. **System Testing:** After integration of each module whether that modules satisfies all the specification or not this testing method is applied. The whole system will work with all type of hardware or not is checked here. It is tested under the different platform.

5. **Acceptance Testing** This testing is done by end user. The client check whether the developed software fulfilled all the requirement which had agreement initially.

1.1.4 Software Testing Techniques

Basically the software testing has been divided into two categories. One is White-Box Testing and other one is Black-Box Testing.

1. **White-Box Testing**: In this type of testing everything has been shown that is way it is called as glass box Testing. Here tester is worried about to find the paths from the code and induce the output. It is based on source code before integration. It applies in every level of testing expect user level. The basic objective is to map which line of code will produce which correct output.

2. **Black-Box Testing:** This testing is a technique for programming testing that inspects the usefulness of an application without peering into its inside structures or workings. This technique for test can be connected to basically every level of programming testing: unit, reconciliation, framework and acknowledgment.Particular information of the application's code/inward structure and programming learning when all is said in done is not needed. The analyzer is mindful of what the product should do however is not mindful of how it does it. Case in point, the analyzer is mindful that a specific info gives back a certain, perpetual yield however is not mindful of how the product creates the yield in any case

   There are different criteria for white box testing are there are . one example is given below.

```
import java.io.*;
public class greatest{
public static void main(string[] args) throws IOException
{
 int m,n,i;
 Scanner in= new Scanner(System.in);
m= in.nextInt();
 n= in.nextInt();
  for(i=0;m!=n;i++)
  {
     if(m>n)
      m=m-n;
     else
     n=n-m;
  }
  System.out.println(m);
  System.out.println(n);
  return 0;
}
}
```

Figure 1.1: An Example Program

- **Statement Coverage :** In this type of coverage each and every statement of the code is covered at least once. It covers all when the condition became true. It cover different flow pf path and also reported whether that path is cover or not. To cover all the statements it require many test cases.Let us consider Figure1.1 Case 1: Suppose in the example m=n=t, where t' is a real number. case 2: m=t and n=t' , where t and

t' are real numbers. Suppose case 1 false the some part of the code will not cover. If Case 2 occur then the loop part will be execute. statement coverage = No. of statements covered/ Total no. of statements.

- **Branch Coverage**: Here the decision node is taken as consideration. In the decision node whether the condition taken is false or true is considered.according our example the test cases are Case 1- m=n, Case 2- m>n, Case 3- m<n. It is stronger coverage than the previous one. So it requires more no. of test cases to cover 100 % coverage.
- **Modified Condition / Decision Coverage**: This technique improves the decision coverage criteria through each and every independently affected condition.e.g If (X && Y) , for this expression test cases are X=True, and Y=False, X=True and Y=True, X=False and Y=True, X=False and Y=False.
- **Multiple Condition Coverage**: All possible out come of the decision is taken here. It is the vigorous technique among the all coverage methods. Here each and every point of entry is executed at least once. For example if && results false then no need to go ahead , if OR result false then it will check further conditions.

1.2 Problem Statement

This section shows the overview of our work. First, automated testing is discussed and then the objective of our proposed approach is discussed.

1.2.1 Automated Testing

Automated testing is done with the help of a software or any tool.Testing action spared around 40% to 50% of the general programming improvement exertion. Now days Testing is the major area of industry. This is utilized as a part of execution

testing, load testing, system testing and security testing,regression testing. In previous days the manual testing was felt due to enormous number of path present in the program. So manual testing unable to detect the error. It is not an repetitive method but some times it adds or replicate new statements. In this way, it is doubtful for a test designer to physically make enough experiments to recognize unobtrusive bugs in all the conceivable execution ways. It is actually a big challenge to produce a test suite that covers all distinctive ways in a robotized way.

1.2.2 Objective of Our Work

The main objective is to achieve more branch coverage as well as MC/DC coverage for Java program. Here we use Concolic testing method which is a symbolic and concrete testing.

The aim of our work is to generate automatic test cases according to branch coverage criteria for Java. Our goal is to cover maximum area with the help of minimum test cases.

- Our objective is to increase MC/DC test data coverage by using code transformer in order to transform program code into logics called transformed program.
- transformer to achieve more branch coverage.
- to use Concolic tester to generate more number of test cases.

1.3 Organization of Thesis

The rest part of the thesis is organized as follows:

1. Chapter 2: In this chapter we discussed the backgrounds of testing with some basic concepts used in our research work, MC/DC Testing and Concolic Testing.

2. Chapter 3: In this chapter we present the literature review where we have described some existing works on automated testing and symbolic testing , Concolic Testing.

3. Chapter 4: In this chapter In this chapter we have discussed about Java Program Code Transformer (JPCT) with working principle and advantages.

4. Chapter 5: In this chapter we discussed about conclusion of our work and as some future extension of our research work.

Chapter 2

Background Details

In this chapter we will discuss some basic definitions of MC/DC testing which will help to understand details of this chapter. Again we will discuss the concepts of Concolic testing and MC/DC coverages.

2.1 Basic concepts

1. **Condition**: One boolean statement with no boolean operator like AND(&&), OR(||) and XOR is called condition. It is an atomic expression that can not be divided into further.

2. **Decision:** A boolean expression with zero or more boolean operator is called decision(predicate). If decisions have zero boolean operator is called as condition.

3. **Group Condition :** Boolean expression comprising of two or more conditions and one or more operators is known as group condition.
   Example: expression ((P==1)&&(Q>=5))||((S>10)&& (T<4)). Here four separate conditions are in this i.e(P==1),(Q>=5),(S>10),(T<4).

2.2 MC/DC Coverage

MC/DC is the abbreviate name of Modified Condition / Decision Coverage. It enhances the coverage by introducing individual effect to each and every condition. The fundamental motivation behind such type of testing is that in the application code every last condition in a choice articulation influences the result of the announcement.

- It invokes every entry and exit point.
- Every decision tries each conceivable result.
- Every condition resides in decisions is exercised with all possible outcomes.
- Every condition present in decision independently influence the outcome of the decision.

2.2.1 Logic Gate:

All logical functions are implemented using Logic Gates. AND gate, NOT gate , OR gate and XOR gate are the basic gates.

AND gate:

We are testing here for n input how many minimum no. of test cases are required . for 4 input AND gate MC/DC criteria are given below.

- For AND gate if all the inputs are true then output is true else not.
- By changing one condition to false then total out put became false.It shows the independent effect of each condition how it affects the overall outcome.

Table 2.1 shows the minimum no. of test cases for 4 input AND gate. For four input the total number of test cases are 16 which is shown in table 2.1

Table 2.1: Test cases for 4 input AND gate

No. of Test Case	Input P	Input Q	Input R	Input S	Output
1	T	T	T	T	T
2	T	T	T	F	F
3	T	T	F	T	F
4	T	T	F	F	F
5	T	F	T	T	F
6	T	F	T	F	F
7	T	F	F	T	F
8	T	F	F	F	F
9	F	T	T	T	F
10	F	T	T	F	F
11	F	T	F	T	F
12	F	T	F	F	F
13	F	F	T	T	F
14	F	F	T	F	F
15	F	F	F	T	F
16	F	F	F	F	F

Table 2.2: Test cases for 4 input OR gate

No. of Test Case	Input P	Input Q	Input R	Input S	Output
1	F	F	F	F	F
2	T	F	F	F	T
3	F	T	F	F	T
4	F	F	T	F	T
5	F	F	F	T	T

OR gate:

For OR gate if all the conditions are false then the total out put became false otherwise if one of the condition is true then the total output became true.

- Here by changing one by one input to true then check how the total out put is being affected. For 'n' number of input n+1 number of output is generated.

Table 2.2 shows the minimum no. of test cases for 4 input OR gate.

2.3 Concolic Testing

Concolic testing is a crossover methodology to programming verification that consolidates Symbolic Execution, which includes speaking to program variables as far as typical variable what's more Concrete Execution, running program on specific inputs. This testing idea was rest presented for way scope. In this procedure, rest of the system is controlled by introducing typical variables with irregular inputs and the way condition is obtained alongside typical execution done on the way acquired. New ways is guided from the past way by flipping or nullifying last condition seen. Figure 2.1 is an example to explain Concolic testing

```
import java.io.*;
public class Grade{
public static void main(string[] args) throws IOException
{
System.out.println("Enter_the_Mark_of_students");
BufferedReader br = new BufferedReader(new  InputStreamReader(System.in));
String ip = br.readLine();
Int testscore = Integer.parseInt(ip);
char grade;
if(testscore>=90){
grade = 'A';
}
else if(testscore>=80){
grade = 'B';
}
else if(testscore>=70){
grade = 'C';
}
else if(testscore>=60){
grade = 'D';
}
else {
grade = 'F';
}
System.out.println("Grade=" + grade );
 }
}
```

Figure 2.1: An Example Program

Let us consider the example program given in Fig 2.1. It is an example of calculation of grade which checks the mark of the student and assign the grade to student. For explaining the Concolic testing we initiate with random inputs to the program. The execution of program begins with secure mark sparing both concrete and symbolic values of variables in the executing way. For entering the first path we enter mark as 91 which is a 'A' grade that is way line number 10 executed and the

current path position is acquired for current execution path is:

$$(testscore >= 90) \tag{2.1}$$

for executing new path the first condition must violate. new constraint created In the case above, is:

$$(testscore >= 80) \tag{2.2}$$

The solver will examine every constraint to check for appropriate path for the program or not.If it is true then new set of values are created for symbolic group and it runs the program with this values. The new path condition acquired by executing system is as per the following:

$$\neg(testscore >= 90) \tag{2.3}$$

The presence of old branch combined with new branches to from a new path.

$$(testscore > 70) \wedge \neg(testscore >= 90) \tag{2.4}$$

The above process will continue till all the conditions will be evaluated.

2.3.1 Concolic Testing Procedure

The whole Concolic testing procedure is carried out in six steps. The steps are given below.

1. **Identification of Symbolic Variable:** From the beginning of the program tester has to choose the symbolic variable that is needed to generate test cases. These variables is chosen from user input, function parameter etc.

2. **Instrumentation of Code:** In this method an additional code is instrumented with original code. It will helpful to maintain a track of a symbolic execution from concrete execution at the time of object code execution.

3. **Concrete Execution :** Instrumented code generated from the above step is compiled and run with a given set of input values. Primarily the instrumented code is executed with set of random inputs. For subsequent step onwards test value is taken from a directed path conditions which is getting from the previous run.

4. **Symbolic Path Execution:** Each symbolic execution path has been gathered by Concolic testing method in symbolic execution module. At each branch point it slams into concrete path execution. When an assignment statement has been found in the time of program execution, It is instrumented statically along with probes. Whenever an if statement found the program variable will be represented in a form of function of symbolic variables or gathering of symbolic path conditions.

5. **Generation of Symbolic path constraint for next iteration:** The symbolic path created from the last step is assisted to nullify one of the last condition and will generate a new constraint for which a feasible path os to be explored in that program.

6. **Generation of Test Input Value:** The model generated by constraint solver will check the feasibility of path constraint which was generated from previous step (step 5).

2.4 Summary

In this chapter, we have discussed briefly about some basic concepts used in our proposed model. Also we analyze some definitions. Briefly we discussed about Concolic testing and its procedure of execution. Apart from we converse regarding some logic gates used in our work.

Chapter 3

Literature Review

Concolic testing and Coverage based testing are the most valuable testing now a days. Many Researchers also used this type of testing in defense application.

3.1 Automated Testing for branch coverage

In software testing field for structural testing automated test data generation is a important feature. There are different types of structural testing are there such Concolic testing, random testing, search based testing etc.

1. **Random testing** A simple strategy for automated testing is portrayed by Godefroid [19] On the off chance that the specialized importance contrasts irregular with orderly, it is in the feeling that vacillations in physical estimations are irregular (erratic or clamorous) versus efficient (causal or legitimate). Random testing generally covers less code coverage.

2. **Search based testing** In this testing the way in which test data is generated is identical to search based optimization problem. Evolutionary technique is based on Genetic algorithm(GA) and Hill climbing method which will helpful to achieve high branch coverage.

3. **Concolic testing** Kim et al. [7] proposed a technique that combines symbolic execution as well as dynamic execution for automatic generation of test cases for path based coverage called as Concolic testing. In our research work we used concolc tester known as JCUTE for Java program.

Godboley et al [3]. proposed the approach of program code transformation technique for C programs to measure MC/DC %. They have used concolic tester CREST to generate test cases and designed coverage analyzer to measure MC/DC percentage. In our approach we are measuring branch coverage percentage for Java program using JCUTE. We developed Java program code transformer based on program code transformer. Our approach gets all benefits of feature of Java programing.

Sen et al. [14] proposed JCUTE to measure branch coverage percentage. In our approach we used JCUTE to generate test cases and measure branch coverage percentage. JPCT is integrated to JCUTE to get increase in branch coverage percentage.

Bokil at al. [9] is a researcher of computer science who developed a tool named as AutoGen that generates test data automatically for C programming that is helpful to reduce cost, as well as effort, . The AutoGen takes the C code and a paradigm such as decision coverage, statement coverage or Modified Condition/Decision Coverage (MC/DC) as info and produces non-repetitive test information that fulfills the specified foundation. We are working under Concolic testing.

Das et al. [1] proposed an augmentation technique of MC/DC test case generation. This methodology deals with automatic generation of MC/DC test suite for C programs. Authors have proposed approach by presenting Boolean Code Transformer (BCT). BCT based on Karnaugh map minimization technique. In our work we developed JPCT which is based on Quine McMluskey minimization

technique which is more powerful technique than K-map.

Tiwari [5] has also done for generating test cases automatically by concolic tester. She used short circuit approach for finding conditions inside decision. But we followed predicate identifier step of JPCT to evaluate condition within decision.

Symbolic execution and program testing - a way to deal with testing was proposed by James c.king [21]. In the methodology, as opposed to passing solid values symbolic values - variables with a few values are breezed through to the project under test. The point of interest of methodology is one symbolic execution speaks to substantial arrangement of concrete execution.

We did our survey with other twelve related works. Table 3.1 represents characteristics of the different approaches such as test cases, coverage type, and computation time.

Table 3.1: Characteristics of different approaches on concolic and coverage based testing

S.No	Authors	Generated Test Cases	Measuring % Coverage	Determined Time Constraints
1	Das et al. [1]	✓	✓	∗
2	Bokil et al. [9]	✓	∗	✓
3	Kim et al. [7] [8]	✓	∗	∗
4	Majumdar et al. [15]	✓	∗	∗
5	Godboley et al. [3] [6]	✓	✓	∗
6	Godboley et al. [3] [18]	✓	✓	∗
7	Burnim et al. [13]	✓	∗	∗
8	Kim et al. [7]	✓	∗	∗
9	Kim et al. [11]	✓	✓	✓
10	Kim et al. [8]	✓	✓	∗
11	Godboley et al. [3] [16]	✓	✓	✓
12	Sen et al. [2] [10] [14]	✓	✓	✓

Chapter 4

Java Program Code Transformer (JPCT)

In this chapter we explain the details of our proposed methodology of work. which consists of two module. In chapter 4 we described the details of one module.

Figure 4.1 shows our proposed framework.

The framework contains two main modules:

1. Java Program Code Transformer (JPCT)
2. Concolic Tester (JCUTE)

In our proposed framework Java Program Code Transformer (JPCT)takes Java program as a input and produced transformed programs as an output. Transformed program is same as original Java program but it does little modification by adding nested if..else statements in side the code. By addition of nested if... else statement it increases the branch coverage of the program without variation of the program output.

This transformed program is feed input to Java Concolic tester (JCUTE). It is a tool for Concolic testing that generates test cases and calculate the percentage of

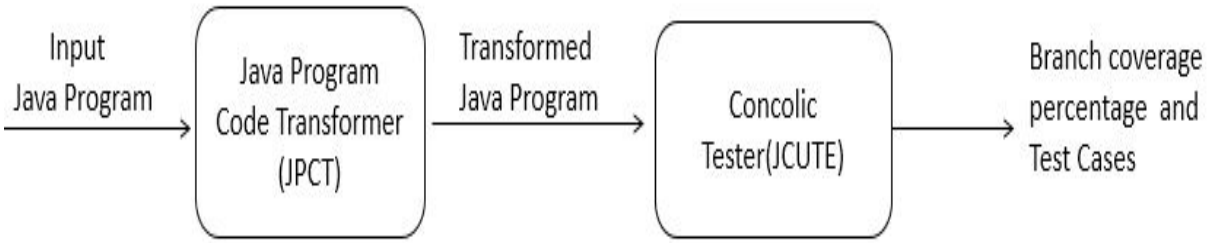


Figure 4.1: Proposed Framework

branch coverage of both original programs as well as the transformed program. It also generated the number of test cases.

4.1 JPCT

It basically one transformation technique.The Java Program Code Transformer uses transformation technique to instrument the Java programs by augmenting it with additional nested if-else conditional statements. This augmentation of code with additional statements causes branch coverage to vary. JPCT modules is based on the fact that by asserting the empty nested if-else conditional statement branch coverage will increase.

JPCT mainly consists of four steps such as

1. Identification of predicates
2. Generation of sum of product form
3. Minimization of SOP using Quine-McMluskey Method
4. Generation of empty nested if-else conditional statements.

In predicate identification phase the program will scan line by lines and it will list out all the predicates in a list and for each predicate generate statements.

Example is shown below Figure 4.2

```
Import java.util.*;
Class PrimeNumbers{
Public static void main(String [] args)
Int n, status=1, num=3;
Scanner in = new Scanner(System.in);
System.out.println( Enter the numbers of prime number
 u want     );
n=in.nextInt();
If(n>=1){
System.out.println(  first +n+ prime numbers are     );
System.out.println(2); }
for(int count=2 ; count<=n;){
for(int j=2;j=sqrt(num);j++)
{
If(num%j==0){
Status=0;break; }
}
If(status!=0)
{
System.out.println(num);
count++;}
Status =1; num++; }
}
}
```

Figure 4.2: Example Program for predicate identification

The above program in the input program and the predicates are listed out by programmatically in 4.3

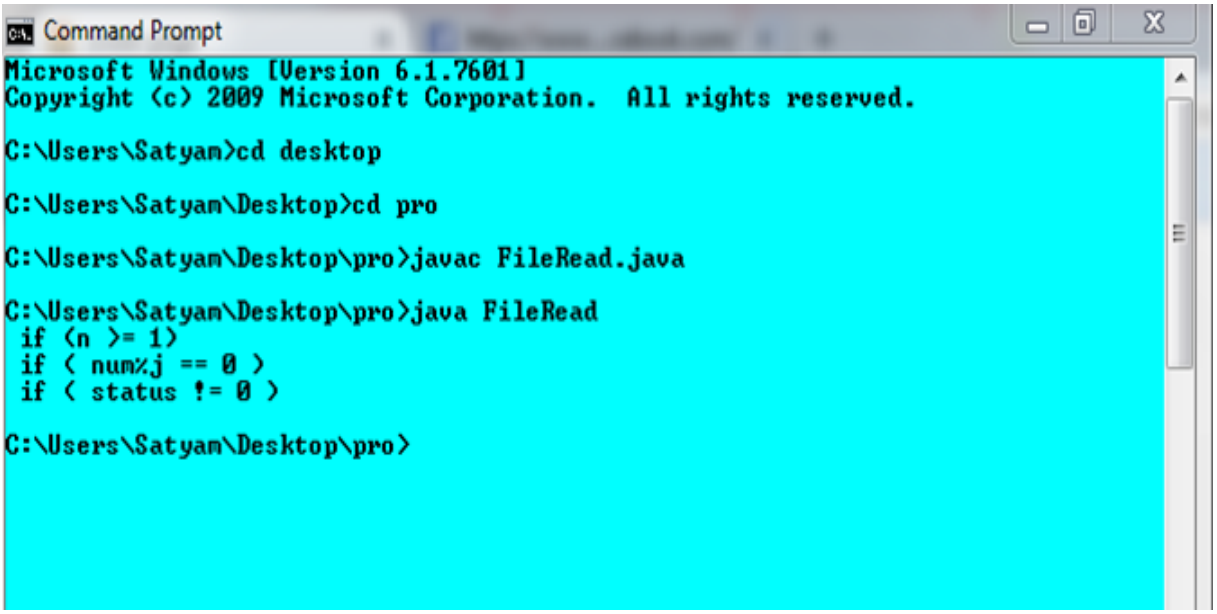


Figure 4.3: No. of Predicate Identified

Predicate identifier does the conversion of an input Java program into standard sum of product form (SOP) using Boolean algebra.

After this building design use QUINE-Mc-MLUSKY Technique for minimization of sum of product. Then this statement is stifled into simple conditions having empty true and false branches which is interpolated just above the predicate. The reason of adding such empty true and false branches is to eliminate duplicate statement execution as the real predicate and the announcements in its branches are held in the program during transformation.

It is a simple method to keep functional equivalence of the code and produces extra test cases for achieving high branch coverage. Hence, JPCT takes Java program as an input and finally produces transformed Java program as its output.

We will represent the schematic diagram of JPCT in below Figure 4.4

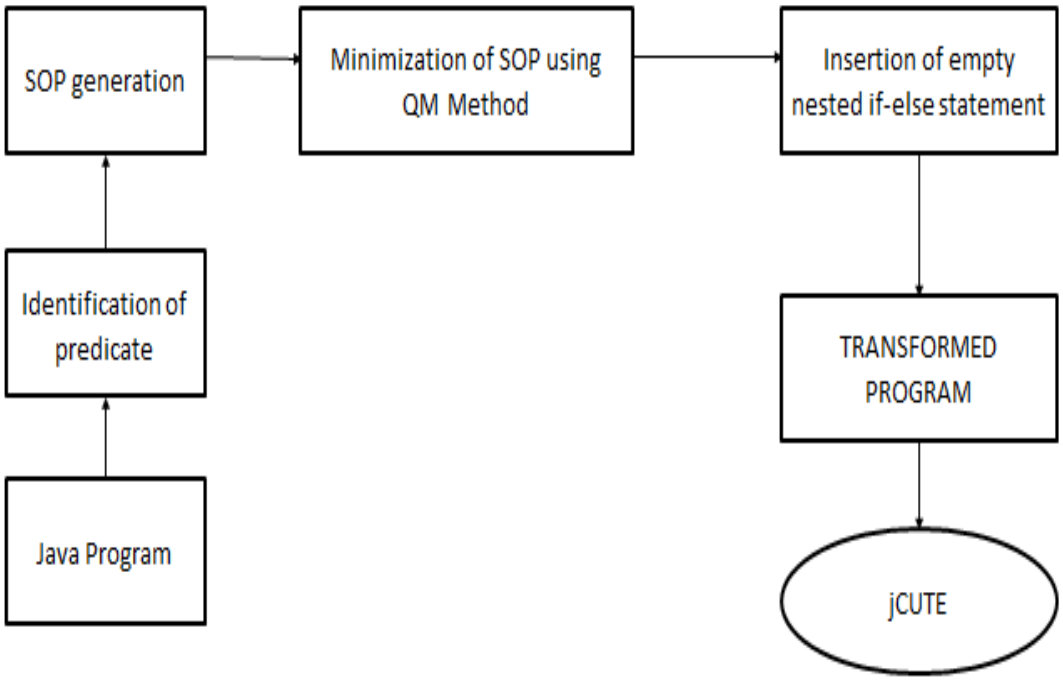


Figure 4.4: Schematic Representation of JPCT

4.1.1 Proposed Algorithm Used in JPCT

For Implementing JPCT in several phases so many algorithms have been required these are explained below.

Algorithm1 for JPCT

First algorithm will convert original program into it's transformed form which has the capacity to cover maximum coverage. The java program code transformer is explained in Algorithm1 with pseudo code. It also include another two sub algorithms namely Algorithm1 and Algorithm2.

1. Identification of Predicates: The motto of this step(line 1 to 5 in Algorithm 1) is to identify all the predicate in Java program. The rationale says that wherever boolean administrator discovered while scanning the Java program character to charcater, duplicate the recognized LOC in some other .text document.

This procedure will execute until all predicates identified.

2. Sum of product(SOP) Generation: In this step, the identified predicate should be simplified. In our algorithm1 each predicate will execute in line number 6 and the SOP will generate in line number 7.

3. Minimization of SOP utilizing Quine-McCluskey: In our algorithm1 line number 8 enumerates minterm for SOP. In line 9 calls the Algorithm2 for minimization of SOP.

4. Insertion of nested empty if-else statements: Line number 10 of algorithm1 invokes Algorithm3 for insertion of empty nested if...else statements. Lastly in line 11 combines all the previous statements and the original Java program.

The output of this algorithm1 is transformed program form original Java program. The algorithm1 is explained below.

Algorithm1: JPCT: Java Program Code Transformer.

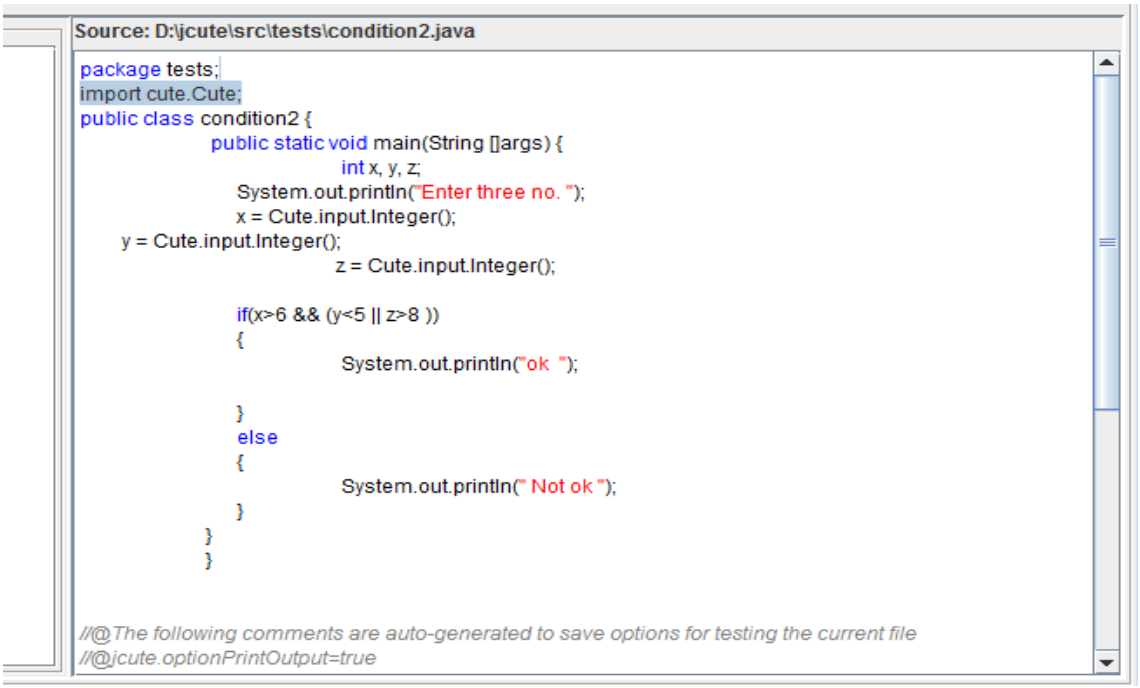
Input: J // Program J is input program

Output: J' // program J' is transformed version

Begin

1: **for** every statement S ∈ J **do**
2: **if** ||or && or unary ! detects in S **then**
3: Identified Predicates ← Save_Record(S)
4: **end if**
5: **end for**
6: **for** every predicate p ∈ Identified predicate **do**
7: Predicate_SOP ← SOP(p) //Generating SOP
8: Predicate_Minterm ← Minterm(Predicate_SOP)
9: Predicate_Minimized ← Minimization_QM(Predicate_Minterm)
10: List_Statement ← Additional_if-else_for_JPCT(Predicate_Minimized)
11: J' ← insert_code(Recorded_Statement,J)
12: **end for**
13: Exit

The original program is shown in Figure 4.5 and the transformed Java program is shown in Figure 4.6


Source: D:\jcute\src\tests\condition2.java

```
package tests;
import cute.Cute;
public class condition2 {
        public static void main(String []args) {
                int x, y, z;
            System.out.println("Enter three no. ");
            x = Cute.input.Integer();
    y = Cute.input.Integer();
                z = Cute.input.Integer();

            if(x>6 && (y<5 || z>8 ))
            {
                System.out.println("ok  ");

            }
            else
            {
                System.out.println(" Not ok ");
            }
        }
        }


//@The following comments are auto-generated to save options for testing the current file
//@jcute.optionPrintOutput=true
```

Figure 4.5: Original Java Program

```
package tests;
import cute.Cute;
public class condition1 {
        public static void main(String []args) {
                int x, y, z;
          System.out.println("Enter three no. ");
          x = Cute.input.Integer();
    y = Cute.input.Integer();
                z = Cute.input.Integer();

          if(x>6)
          {
                if(y<5){}
                else {}
          }
                else {}
          if(x>6)
          {
                if(z>8){}
                else {}
          }
                else {}

          if(x>6 && (y<5 || z>8 ))
          {
```

Figure 4.6: Transformed Java Program

In Algorithm2 we used Quine-McClusky Technique for minimization of identified predicate. This technique is more powerful than the K-map.The input of this algorithm is computed predicate and output is minimized form of the computed predicate. The line 1 to line 3 execute for every minterm to get changed over into binary form. Line 4 sorts the computed A_record according to the presence of 1s in binary number and maintain it in B_record.The line 5 to 16 will execute for every bit form first to last for every record. Line no. 8 calculates the 1 bit difference between current group and further groups. The compared and uncompared group are distinguished by line 10 to line 13. Line 17 and line 18 figures Prime Implicant and Prime Implicant Essential separately. The result of Predicate_minimized id formed in line 19 by PETRIC method and line 20 of this Algorithm2 returns the value of minimized form to the Algorithm1.

The Algorithm2 is written below .

Algorithm2: Minimization_QM.

Input:Predicate_Minterm

Output:Predicate_Minimized

Begin

1: **for** each Minterm M ∈ PredicateMinterm **do**
2: A_record ← Conversion into Binary(M)
3: **end for**
4: B_record ← Sort(A_record) //According to the number of one in each binary number
5: **for** each record l ∈ B_record **do**
6: **for** every first_group to last_group ∈ Groups **do**
7: **for** every bit in group ∈ total bits in group **do**
8: differnce_of_1_bit ← Compare(current_group,next_group)
9: **end for**
10: **if** differnce_of_1_bit==1 && remain_real_dash_location **then**
11: bit will reinstated with – and insert checkcharacter "X "
12: **else**
13: insert check character ∗ for incomparable group
14: **end if**
15: **end for**
16: **end for**
17: Prime_Implicant ← incomparable with other and revealed with ∗
18: Essential_Prime_Implicant ← Coverage_Table(Prime_Implicant,Minterms)
19: Predicate_Minimized ← Assignment_variables and Compliment_variables for testing Prime_Implicant //According to PATRIC METHOD
20: return Predicate_Minimized

In Algorithm 3, the input is Predicate minimized and output is recorded statement. The line 1 to line 13 execute for each condition in group of condition for each && connected conditon group in Predicate_ Minimized. Line 3 to line 10 execute to make if statement and maintain it in Recorded statement. Line 11 and line 12 generating an empty false branch for the anterior condition and maintaining it in Recorded statement. Line 4,7 are repeated by line 13 and 14 and 8 for each condition that belong Predicate_Minimized and not belonging to any condition group. Line 17 to line 21 execute when the Predicate_Minimized is in else-if form. That is why, Algorithm 3 returns the last Recorded statement to Algorithm 1. The Algorithm3 is shown below.

Algorithm 3: Additional_if-else_for_JPCT.

Input: Predicate_Minimized

Output: Recorded_statement

Begin

1: **for** every && connected condition_grp∈ Predicate_Minimized **do**
2: **for** every condition C ∈ condition_group **do**
3: **if** C is the first_condition **then**
4: generate an if statement S having C as the condition
5: Recorded_Statement ← save_record(S)
6: **else**
7: generate a nested if statement S with C as the condition generate an empty True_branch TR_b and an empty False_branch FL_b in order
8: Recorded_Statement ← save_record(strcat(S,TR_b,FL_b))
9: **end if**
10: **end for**
11: create an empty False_branch FL_b for initial condition
12: Recorded_statement ← save_record(FLb)
13: **end for**
14: **for** every condition ∈ Predicate_Minimized and ∉ any condition_group **do**
15: repeat lineNo. 4, 7,8
16: **end for**
17: **if** Predicate_Minimized is an else-if predicate **then**
18: create an if(false) statement S
19: create an empty True_branch TR_b
20: Recorded_Statement ← save_record(strcat(S,TR_b))
21: **end if**
22: return Recorded_statement

4.2 JCUTE

It is a concolic testing tool for Java programs. It consists of two module such as instrumentation module and library module. These modules are used for symbolic execution, for controlling thread scheduling and to solve constraints. The instrumentation modules inserts code in the program under test so that the instrumental programs call the library at runtime for performing symbolic execution. JCUTE is a tool with all graphical features. For the instrumentation of Java programs JCUTE used CIL and SOOT compiler. A semaphore is associated with JCUTE for it's instrumentation. JCUTE will add an operation to these semaphore for instrumentation before each shared memory access. These semaphores are helpful to control thread scheduling at run time. JCUTE also used for solving arithmetic inequalities for which the library module is useful. For linear programming problem JCUTE saves all the input values and schedules in a file system. As such the users of JCUTE can replay the program to reproduce the bugs. The debugger also used for replay of the program. For sequential programs JCUTE generates JUNIT test cases. Due to the GUI features the multi threaded execution also visualized to programmer.

It is also used for calculating branch percentage coverage. Here in our experiments we follow two approaches one is with the help of code transformation technique and other is without help of code transformation technique. In the first technique the original Java program is used to calculate the branch coverage with the help of JCUTE only. In second technique the transformed program is used to calculate the branch coverage with the help of JCUTE and JPCT for achieving high branch coverage. The cause of achieving high branch coverage the augmentation of additional empty if..else statements for a target Java programs. The changed form of the target Java program is called Transformed Java Program. The output of the programs can not affected by transformation technique. After the instrumentation of Java program is over the additional conditional statements inserted earlier for transformation are removed from the code.

4.2.1 Procedure to Calculate Branch Coverage Difference

In the below we will describe the detailed procedure for calculation of branch coverage percentage difference between original and transformed program. The steps are given below.

Step1: ***Generate Test_Suite1 for target program J***

JCUTE is used for generation of Test Suite1. JCUTE takes Java program J as an input and generate test cases. It will take random number as an input at first and schedule it which specifies the order of execution of thread. Then the algorithm executes the code with the generated input and the schedule.

Step2: ***Calculate Branch_Coverage_1 percentage***

To calculate Branch_Coverage_1 percentage, we use JCUTE. This algorithm takes the target program J along with the generated test cases Test Suite1 as input and computes the Branch_Coverage_1 percentage as output. The algorithm identifies all the predicates present in the program and applies each of the test cases for each condition of the predicate to compute the Branch Coverage percentage. The Coverage Analyzer calculates the branch coverage percentage.

Step3: ***Transform J to J'***

In this step, the target Java program J is transformed into a transformed Java program J by using Java Program Code Transformer module(JPCT). The transformed Java program contains additional nested empty if-else conditional statements. The JPCT algorithm in turn generates these additional conditional statements.

Step4: ***Generate Test Suite2 for J'***

This step is like that of Step 1. At the same time, not at all like in Step 1, the tool accepts the transformed program J as input to create the obliged test cases as output.

Step5: ***Calculate Branch_Coverage_2 Percentage***

This step is like that of Step 2. This step is like that of Step 2. At the same time, not at all like in Step 2, the Coverage Analyzer takes changed Java program J alongside

the created Test Suite2 as input to figure Branch Coverage 2 % as output.

Step6: ***Compare Branch_Coverage_1 percentage and Branch_Coverage_2 percentage***

This step compares the Branch Coverage percentages computed in Step 2 and Step 5, and computes their difference as given below.

$$\boxed{Difference = Branch\_Coverage\_2\% - Branch\_Coverage\_1\%}$$

4.3 Experimental Analysis

We done our experiment using ten benchmark Java programs taken from open source laboratory and some are student assignments. The specification of all the ten programs are given in Table 4.1. The attribute of our given Table which are written in (') are transformed program and which are written without (') form are original Java programs. For example in column 3 of Table 4.1 LOC is size of input program and in column 4 LOC' is the size of transformed Java program.

Table 4.1: Characteristics of the Different Experimental Programs

SL. No.	Program Name	LOC	*LOC′*	No. of function invoked	No. of classes	No. of predicate	Total No. Branches	No. of Variables
1	Condition	21	32	1	1	2	15	3
2	Weight	24	42	1	2	3	20	4
3	Quick Sort	68	76	3	4	2	18	8
4	Nonce	286	350	10	11	25	146	14
5	StringBuffer	421	466	8	4	17	56	6
6	SwitchTest	59	71	2	3	4	15	5
7	Producer Consumer	20	27	4	2	2	8	4
8	BS Tree	255	297	3	5	6	28	8
9	Array Sort	27	33	2	2	2	15	3
10	Static InstanceProblem	24	32	1	1	1	5	2

4.3.1 Comparison of Branch Coverage Percentage

The number of branch covered of all ten experimented programs are shown in Table 4.2. From Table 4.2 it is observed that the number of branch covered by JPCT+JUTE (transformed program) is more than the JCUTE (normal input program) only.

In Table 4.3 the branch coverage percentage achievement of all ten programs by both JCUTE AND JPCT+JCUTE are given. The branch coverage variance of both JCUTE and JPCT are shown in last column (column 5) of Table 4.3. Consequently, from analyzing in Table 4.3 demonstrate that we accomplish 17.21% of normal increment in branch coverage for each of the ten experiment programs. The average branch covered of JCUTE is 57.17% (input Java program) and JCUTE+JCPT (transformed program) is 74.38%.

We also represent all percentage coverage in graphical form in figure 4.7 . In Figure 4.7 x-axis represent all then experimented programs and in y-axis is represented as percentage of branch covered. The difference of average branch coverage of JCUTE(input program) and JCUTE+JPCT (transformed program) is shown in Figure 4.8

Table 4.2: Number of branches covered

SL No.	PROGRAM NAME	Branch Covered (JCUTE)	Branch Covered (JPCT+JCUTE)
1	Condition	6	14
2	Weight	8	15
3	Quick Sort	15	15
4	Nonce	89	106
5	String Buffer	33	36
6	SwitchTest	12	13
7	Producer Consumer	5	6
8	BS Tree	19	23
9	Array Sort	9	11
10	Static InstanceProblem	3	4

Table 4.3: Analysis of Branch Coverage Percentage

SL	Program	Branch Coverage %(jCUTE)	Branch Coverage %(JPCT+jCUTE)	Variance%
1	Condition	40%	93%	53%
2	Weight	40%	75%	35%
3	Quick Sort	83.33%	88.88%	5.55%
4	Nonce	60.95%	72.60%	11.65%
5	String Buffer	58.92%	60%	1.08%
6	SwitchTest	75%	84.18%	9.18 %
7	Producer Consumer	60%	72.50%	12.50%
8	BS Tree	61.90%	76.19%	14.29%
9	Array Sort	42.85%	61.42%	18.57%
10	Static InstanceProblem	48.74%	60%	11.26%

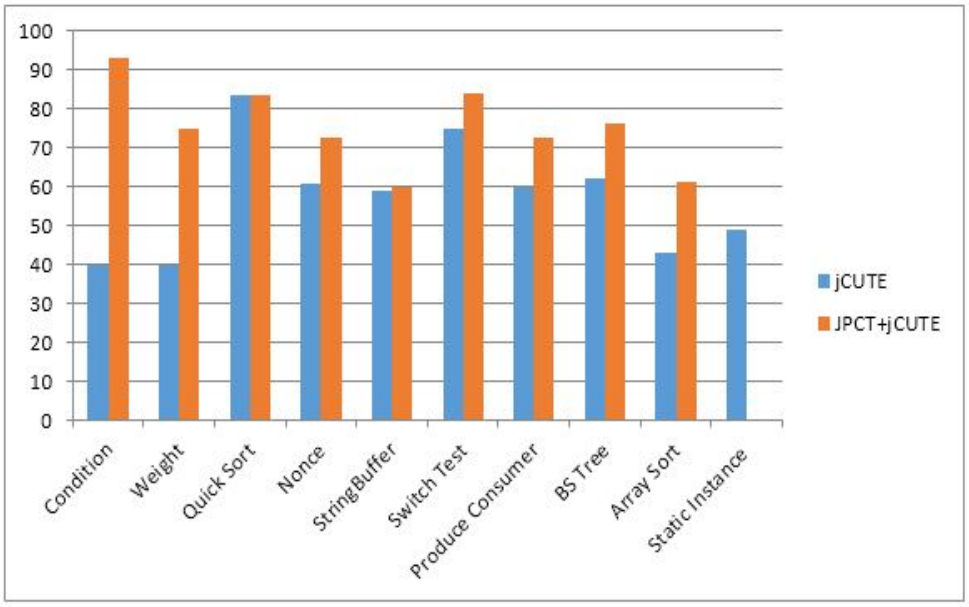


Figure 4.7: Bar Graph For Percentage Branch Coverage Achieved

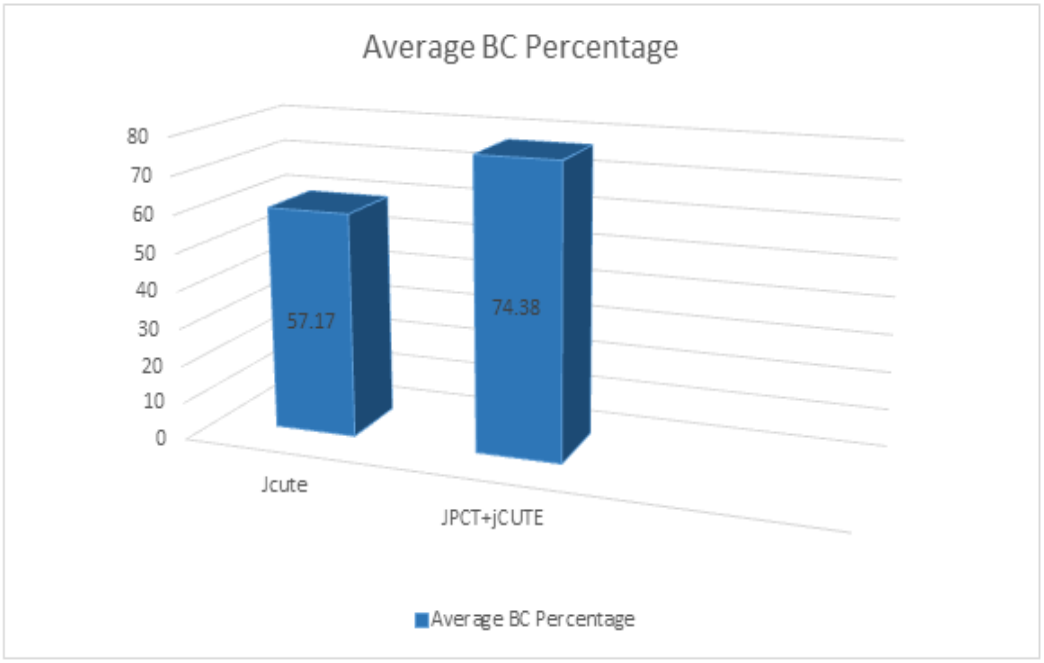


Figure 4.8: Average BC% for Different Techniques

4.3.2 Analysis of Number of Test Cases Generated

The test case helpful to coverage more and also to provide better quality product without any bug to user. In our transformed program is able to generate more number of test cases as compare to original program without affect of the output of the programs.

Number of Test cases generated is more in JPCT+JCUTE (transformed program) than JCUTE (input program) only which is represented in Table 4.4 below. It is also represented in pictorial form in figure 4.9 where x-axis represent name of programs and y-axis represents number of Test cases.

From examining Table 4.4 the transformed program generates more 11.29 % Test cases than normal input programs.

Table 4.4: Number of Generated Test Cases

SL No.	Program NAME	No. of Test Cases (JCUTE)	No. of Test Cases (JPCT+JCUTE)
1	Condition	4	5
2	Weight	5	6
3	Quick Sort	1	1
4	Nonce	24	24
5	String Buffer	8	8
6	SwitchTest	6	7
7	Producer Consumer	1	2
8	BS Tree	6	8
9	Array Sort	4	5
10	Static InstanceProblem	3	3

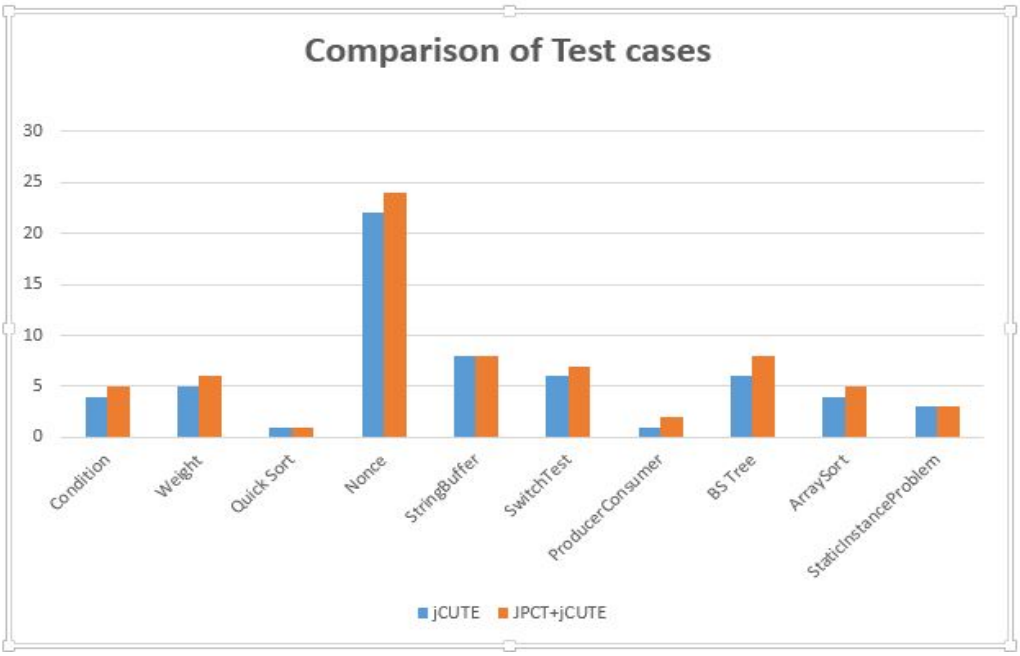


Figure 4.9: Generation of Test Cases

4.3.3 Comparison of Total Number of Path Covered

The path covered fully affect the branch coverage. If the number of path covered is more then the branch coverage is also more. In our experiment the JPCT+JCUTE (Transformed program) covered more path than JCUTE (input Java programs). The JPCT+JCUTE covers 2.46% more path than JCUTE only.

Below table 4.5 is shows the Analysis of total number of paths covered.

Table 4.5: Total Number of Paths Covered

SL No.	Program NAME	No. of Paths (JCUTE)	No. of Paths (JPCT+JCUTE)
1	Condition	5	10
2	Weight	4	5
3	Quick Sort	1	1
4	Nonce	100	100
5	String Buffer	380	380
6	SwitchTest	7	8
7	Producer Consumer	2	3
8	BS Tree	15	18
9	Array Sort	11	13
10	Static InstanceProblem	3	3

4.3.4 Analysis of Computation Time

Time is a major factor of any algorithm depend on which the total speed of the system varies. By little delay of any algorithm if it gives more performance than other algorithms then it is good.

The time analysis of our framework is given in Table 4.6. The third and fourth column of Table 4.6 shows computation time of JCUTE and JPCT respectively. The last column of the Table 4.6 represents computation time of both JPCT+JCUTE (Transformed Program) which is a little bit higher than normal programs.
The average computation time of all ten experimented programs is 28156.9 milliseconds.

Table 4.6: Computation Time for JCUTE and JPCT+JCUTE

SL	Program	Computation Time for JPCT(ms)	Computation Time for JPCT(ms)	Computation Time for JPCT+jCUTE(ms)
1	Condition	1218	1120	2338
2	Weight	1609	1370	2979
3	Quick Sort	328	800	1128
4	Nonce	17725	9590	27315
5	String Buffer	131896	28532	160428
6	SwitchTest	12190	9874	22064
7	Producer Consumer	6098	7396	13494
8	BS Tree	9072	10284	19356
9	Array Sort	4286	6109	10395
10	Static InstanceProblem	12201	9871	22072

4.4 Summary

In this chapter, We have discussed about the Java Program Code Transformer (JPCT). It gives better performance as compare to normal Java program without changing the output of the total programs or system. It achieves high branch coverage percentage than the normal program as well as it generates more number of test cases than the original Java program. It also covers more path. Its computation time is little bit more than normal program with JCUTE, But as compare to overall performance JPCT is better .

Chapter 5

Conclusion and Future Work

We proposed a framework named A novel frame work for improved branch coverage analyzer for test case generation that is based on the coverage analysis of Java programs. We discussed the detailed steps of our proposed approach along with the working principles of the modules (Java Program Code Transformer, and JCUTE) of architectural framework. The experimental results show that the proposed approach of test case generation achieved better branch coverage in comparison to the existing approaches. Our proposed approach achieved 17.21% more branch coverage as compare to previous which rises in BC percentage is achieved with an average computation time of 28156.9 milliseconds.

In future our research work will be extended towards distributed manner. Because in distributed manner the computation time will be less which is the main significant of the real-time system. In future it also possible to develop the program transformer that will be platform independent. Also we add MC/DC coverage to our module.

Dissemination

1. S. Godboley, **A. K. Sahani**, and D. P. Mohapatra, ABCE: A Novel Framework for Improved Branch Coverage, In preceding of 3^{rd} *International Conference on Soft Computing and Software Engineering* Procedia of Elsevier, University of California, Berkley, USA, March 5-6, 2015.(Published)(Chapter 4)

Bibliography

[1] Avijit Das.*Automatic Generation of MC/DC Test Data.* Master Thesis,Computer Science Engineering, Indian Institute of Technology,Kharagpur, India 2012.

[2] K. Sen, D. Marinov, and G. Agha, Cute: a concolic unit testing engine for c," Vol. 30, No. 5, pp. 263-272, ACM.2005.

[3] S. Godboley. Improved modified condition/ decision coverage using code transformation techniques. M.tech Thesis, NIT Rourkela, 2013.

[4] Das. Avijit, and Rajib Mall. *Automatic generation of MC/DC test data.* Journal of software engineering,Vol 3, ACTA press , 2013.

[5] Tiwari, S. . *Automatic Generation of Testcases for High MC/DC Coverage* (M.tech Thesis), Indian Institute of Technology Kanpur, 2014.

[6] S. Godboley, G. Prashanth, D.P. Mohapatra, and B. Majhi. Increase in modified condition/decision coverage using program code transformer. In *IEEE 3rd International Advance Computing Conference (IACC)*, pp. 1400-1407, Feb 2013.

[7] M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing on embedded software: Case studies. *In IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST),* , pp. 390-399 , 2012.

[8] Y. Kim and M. Kim. Score: a scalable concolic testing tool for reliable embedded software. *In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering,* pp. 420-423. ACM, 2011.

[9] P. Bokil, P. Darke, U. Shrotri, and R. Venkatesh. Automatic test data generation for c programs. In *Third IEEE International Conference on, Secure Software Integration and Reliability Improvement, SSIRI pp.* 359-368, July 2009.

[10] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools (tools paper). Technical report, DTIC Document, 2006.

[11] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim. Automated unit testing of large industrial embedded software using concolic testing.*In IEEE/ACM 28th International Conference on, Automated Software Engineering (ASE), pages 519-528*, Nov 2013.

[12] CREST. http://code.google.com/p/crest .

[13] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. *In Proceedings of the 23rd IEEE/ACM international conference on automated software engineering*, pages 443-446. IEEE Computer Society, 2008

[14] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, M. Wermelinger and H. Gall, editors, pages 263-272, 2005.

[15] R. Majumdar and K. Sen. Hybrid concolic testing. *In Proceedings of the 29th International Conference on Software Engineering, ICSE 07*, IEEE Computer Society, pp. 416-426, Washington, DC, USA, 2007.

[16] S. Godboley and D. P. Mohapatra. Time analysis of evaluating coverage percentage for c program using advanced program code transformer. pages 9197. Computer Society of India, 7 th CSI International Conference on Software Engineering, 11 2013.

[17] Dupuy, A., & Leveson, N. . An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In Proceeding of The 19th IEEE Digital Avionics Systems Conference, Proceedings. DASC,Vol. 1, pp. 1B6-1, 2000.

[18] S. Godboley, G. Prashanth, D. Mohapatra, and B. Majhi. Enhanced modified condition/decision coverage using exclusive-nor code transformer. *In International Multi-Conference on Automation, Computing, Communication, Control and Compressed Sensing (iMac4s)* , pp. 524-531, March 2013.

[19] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. SIGPLAN Not., vol. 40, no. 6, pp. 213-223, June 2005

[20] S.Godboley, S . Panda, and D.P Mohapatra, SMCDCT: A Framework for Automated MC/DC Test Case Generation Using Distributed Concolic Testing. *In 11th International Conference on Distributed Computing and Internet Technology.*Lecture Note of Computer Science (LNCS) Springer, pp. 199-202, 2015

[21] King, J. *Symbolic Execution and Program Testing* Testing. Communications of the ACM, ACM press,vol. 7 pp. 385-39, 1976.